APPENDIX - Example Pseudo-code Implementations

The following pseudo-code segments are provided to aid in understanding the various parts of the present invention. They should not be construed as complete or optimal implementations. Note that these codes illustrate the operation of the basic system described above, without any of the additional enhancements discussed. Although given as software code, the actual implementations may be as stored-program(s) used by a processor, as dedicated hardware, or as a combination of the two.

Example Implementation of decoder 156

```
/* Output begin training sync pattern */
for (i=0;i<syncPhaseLength;i++)
        Output maximum code value
for (i=0;i<syncPhaseLength;i++)
        Output minimum code value

/* Output training data */
for (i=0;i<trainRepeat;i++)
        for (j=0;j<trainLength;j++)
                Output training pattern element j

/* Output end training sync pattern */
for (i=0;i<syncPhaseLength;i++)
        Output minimum code value
for (i=0;i<syncPhaseLength;i++)
        Output maximum code value

sum=0
loop forever
        Read an input data byte
        Output data byte
        Convert byte to equivalent linear value, x
        sum += x;
        if (current output sample number is a multiple of dcPeriod)
                if (sum > 1.0)
                        recover = maximum code value
                else if (sum < -1.0)
                        recover = minimum code value
                else
                        recover = code value having linear value closest to -sum


                Output recover
                sum -= linear equivalent of recover
```

Example Implementation of clock synchronizer 260

Initialize filters array to be the impulse response of a low-pass

filter with digital cutoff frequency PI/Nu.

Initialize lpfBuffer array to all zeroes.

```
snum = -lpfLen/2;
lpfPos = 0;

Loop forever
        Read an input sample into val
        /* Store value in circular buffer, lpfBuffer[] */
        lpfBuffer[lpfPos] = val;
        lpfPos = (lpfPos+1)%lpfLen;
        snum++;
        while (snum >= period)
                /* Extract an output  from resampler at 'period' units after
                   the previous extracted sample */
                snum = snum - period;
                phase = (int)(snum*Nu);
                frac = snum*Nu-phase;

                /* Compute output from two adjacent phases of filter */
                                                        lpfOut1 = lpfOut2 = 0;
                for (i=0,p=lpfPos;i<lpfLen;i++,p=(p+1)%lpfLen)
                        lpfOut1 += lpfBuffer[p]*filters[i*Nu+phase];
                        lpfOut2 += lpfBuffer[p]*filters[i*Nu+phase+1];
                /* Interpolate */
                result = lpfOut1*(1-frac)+lpfOut2*frac;

                Write result as an output sample
```

Example Implementation of decoder 156

```
Loop forever
        Read a sample from clock synchronizer into 'samp'
        /* Put samp at the end of 'inBuffer' */
        inBuffer[inPos] = samp;
        inPos = (inPos+1)%inBufLen;

        /* Check if we are just finishing a sync pattern */
        if (last syncLength samples read are all negative and previous
```

```
            syncLength samples are all positive)
            inTraining = 1;
    else if (last syncLength samples read are all positive and previous
    syncLength samples are all negative)
            inTraining = 0;

    /* Add sample to FFE buffer */
    ffeBuffer[ffePos] = samp;
    ffePos = (ffePos+1)%ffeLen;

    /* Only need to compute output every second sample */
    if (ffePos%2 == 0)
            /* Perform FFE equalization */
            ffeOut = DotProd(&ffeBuffer[ffePos],&ffeWts[0],ffeLen-
            ffePos);
            ffeOut += DotProd(&ffeBuffer[0],&ffeWts[ffeLen-
            ffePos],ffePos);

            /* Subtract FBE output */
            ffeOut -= fbeOut;

            /* Convert output to nearest code */
            codeOut = Linear2Code(ffeOut);

    if (inTraining)
            /* Use training pattern to calculate error */
            eEst = ffeOut - Code2Linear(train[tpos])
            tpos = (tpos+1)%trainLength;
    else
            /* Calculate decision feedback error */
            eEst = ffeOut-Code2Linear(codeOut);

    /* Update equalizers */
    for (i=0;i<ffeLen;i++)
            ffeWts[i] += ffeGain*eEst*ffeBuffer[(ffePos+i)%ffeLen];
    for (i=0;i<fbeLen;i++)
            fbeWts[i] += fbeGain*eEst*fbeBuffer[(fbePos+i)%fbeLen];
    /* Calculate derivative of output with respect to time */
    out[0] = out[1];
    out[1] = out[2];
```

```
        out[2] = ffeOut;
        deriv = (out[2]-out[0])/2;

        /* Calculate phase error */
        num *= pllPole;
        denom *= pllPole;
        num += prevEEst*deriv;
        denom += deriv*deriv;
        pdAdjust = num/denom;

        /* Update resampler period (fed to clock synchronizer) */
period = midPeriod+pllGain*pdAdjust;

        /* Save error estimate for next cycle */
        prevEEst = eEst;

        /* Compute next FBE output */
        fbeBuffer[fbePos] = ffeOut;
        fbePos = (fbePos+1)%fbeLen;
        fbeOut = DotProd(&fbeBuffer[fbePos],&fbeWts[0],fbeLen- fbePos);
        fbeOut += DotProd(&fbeBuffer[0],&fbeWts[fbeLen- fbePos],fbePos);

        /* Output a sample (delayed) if we are active */
        if (outputting)
                if (oSampNum>0 && (oSampNum%dcPeriod) != 0)
                        Output outBuffer[outBufPos]
                oSampNum++;

        /* Store new sample in output buffer */
        outBuffer[outBufPos] = codeOut;
        outBufPos = (outBufPos+1)%outBufLen;

        /* Check if sync in buffer and set outputting accordingly */        if
(last syncLength/2 samples placed in outBuffer are negative
                        and previous syncLength/2 samples are positive)
                outputting = 0;
        else if (last syncLength/2 samples placed in outBuffer are
                negative and prev. syncLength/2 samples are positive)
        outputting = 1;
                oSampNum = -syncLength + 1;
```